编译原理实验一 实验报告

组长：白家杨 161220002 组员：段高磊 161220036

一、实验说明

（1）本次实验不改变助教的Makefile文件，编译方法为直接在Code文件夹下使用make指令。想要检测特定的程序文件（比如example.cmm）：先执行make编译，然后执行以下命令：

```
./parse example.cmm
```

（2）选作**任务1.2**：识别指数形式的浮点数

二、词法分析

借助GNU Flex分析工具，我们编写lexical.l作为提供词法规范的正则表达式。参考C--语言的文法定义的Tokens部分，我们需要自行考虑**INT**，**FLOAT**以及**ID**的定义。

（1）对于INT类型正则表达式如下：

```
digit  [0-9]
integer  0|[1-9][0-9]*
```

对于INT类型需要考虑的点就是不能出现0开头的多位数字。

（2）对于FLOAT类型正则表达式如下：

```
FLOAT  (({digit}+\.)|({digit}*\.{digit}+))f?
FLOAT_exp (({digit}+\.)|({digit}*\.{digit}+))[Ee][+-]?{integer}
```

FLOAT的定义可以借助之前digit和integer的定义来书写简化。上面**FLOAT**表示非指数形式的浮点数，**FLOAT_exp**表示指数形式的浮点数。这里考虑浮点数中小数点前后必须有数字出现，比如可以表示“.7”（0.7）、“8.”（8.0）、“6.7f”（6.7），同时我们考虑显式表示方法：FLOAT后缀可以加上f。

对于指数型浮点数，可以拆成FLOAT和指数部分，指数部分一定出现[Ee]。举例可以表示“3.e-4”（$3.0 * 10^{-4}$）、“.77E+3”（$0.77 * 10^{3}$）、“12.48e6”（$12.48 * 10^{6}$）

（3）对于ID类型正则表达式如下：

```
letter  [_a-zA-Z]
ID  {letter}({letter}|{digit})*
```

ID表示的是除去保留字以外的所有标识符，且必须由字母或下划线开头。这里也利用到了前面定义的digit，简化书写。ID的正则表达式可以表示比如“_exp”、“func10”、“compiler_lab_1”等。

（4）此外，我们另外加入了SPACE（空格），TAB（制表符），LINEBREAK（换行或换页）：

```
SPACE   " "
TAB   "\t"|"\v"
LINEBREAK   "\n"|"\r"|"\f"
```

（5）对于剩下的匹配，即任意输入的点规则（“.”），报告错误类型A，同时将变量ErrorLex设为true。ErrorLex是否出现词法错误的标志，初始设为false。即：

```
.    { ErrorLex=true; printf("Error type A at line %d: Mysterious character \"%s\"\n",yylineno,yytext); }
```

在词法分析中，识别到词法单元后，利用yytext可以获得字符属性信息，以及词法单元类型，可以新建一个语法树节点，这作为接下来语法分析和语法树的构建奠定基础。

实现风格及亮点：利用正则表达式的部分可重复使用，极大简化规则表示和后续的更正修改。

三、语法分析

借助GNU Bison工具，我们需要完成包括语法规则等在内的Bison源代码（编写syntax.y）；根据C--语言提供的相关文法定义，我们将 **High-level Definitions**，**Specifiers**，**Declarators**，**Statements**，**Local Definitions** 以及 **Expressions** 下的语法规则写入**syntax.y**文件，并在每条规则后加入执行动作，如将当前的符号关系插入语法树、标记出错等。

1. 语法树结构

当被测试的程序没有错误时，我们需要打印出该程序对应的语法树。根据提供的输出样例，我们定义语法树的节点如下:

```
typedef struct TreeNode
{   int line;
    int depth;
    enum NodeType nodetype;
    char name[33];  // 32+1
    char value[40]; // 39+1
    struct TreeNode* parent;
    struct TreeNode* firstChild;
    struct TreeNode* next;
    struct TreeNode* lastChild;
}TreeNode;
```

其中，**line** 记录终结符或非终结符的所在行号；**depth** 记录当前节点在语法树中的深度，以打印空格维持输出结构；**nodetype** 记录当前节点的类型，非终结符所在的节点以 **TYPE_**开头，终结符以 **TOKEN_** 开头（详细定义见**const.h**文件；或许会便利后续的实验）；**name** 存储符号的名称；**value** 存储符号对应的值；后续四个变量构建树结构。

注：由于float范围为：-3.4E38~3.4E38（十进制最长为39位），且整型数不超过32位、标识符长度不超过32；因为以字符串形式获取词法单元对应的词素，故将value的长度设置为40位

构建的函数如下：

```
TreeNode* newNode(int line,enum NodeType tp,char name[],char value[]); //创建新节点
void insertTree(TreeNode* cur,TreeNode* p); //将当前语法规则对应的符号关系插入语法树
void computeDepth(TreeNode* node); //计算树中每个节点的深度
void printTree(TreeNode* node);  //打印语法树
void deleteTree(TreeNode* root); //销毁语法树
```

2. 细节处理

（1）属性值类型的确定

为了无错误时可以输出语法树，则在分析语法结构的时候就要构建相关语法树，进而我们需要将终结符和非终结符的属性值都定义为树的节点类型，如下：

```
%union{
    struct TreeNode* tree_node;  //declared types
}
%token <tree_node> INT FLOAT ID TYPE  //declared tokens
...
%type <tree_node> Program ExtDefList ExtDef  //declared non-terminals
...
```

（2）语法单元位置的获取

根据实验指导，在Flex源文件添加宏定义 **YY_USER_ACTION**，并在发现换行符时复位变量 **yycolumn**（赋值为1）， 则可以用**@$**、**@1**、**@2** 等获取位置信息。

（3）二义性与冲突处理

- 根据C--语言中运算发的优先级和结合性，规定终结符的结核性（详细见syntax.y）
- 常见的if-else嵌套冲突：定义**%nonassoc LOWER_THAN_ELSE** 以及**%nonassoc ELSE** 并添加入if-else的语法规则中

（4）错误恢复

添加error符号到语法规则中，我们在高层的产生式和底层的产生式都做了一些处理，对于常见的语法错误都能检错输出，如:

```
Def : Specifier error SEMI ; //可以识别常见的变量定义错误，如 int x = 3.e; 等
Stmt :IF error RP Stmt %prec LOWER_THAN_ELSE
    |   IF error RP Stmt ELSE Stmt { ErrorSyn=true; } ; //可以识别if-else语句条件部分的错误
```